SOFTWARE MAINTENANCE: RESEARCH AND PRACTICE, VOL. 8, 309–320 (1996)

Practice

Methods of Measuring Software Reuse for the Prediction of Maintenance Effort

RONALD J. LEACH
Department of Systems and Computer Science, Howard University, Washington, DC 20059, U.S.A.

SUMMARY

A major difficulty in evaluating the costs of reusing software is determining the amount of reused software artefacts in systems. Determining the amount of reuse in a system is important for software maintenance because reused software is likely to need less corrective maintenance than newly developed software. Reusing software can also decrease costs of testing and integration.

In this paper, we describe some practical techniques for measuring the amount of software reuse using simple tools. The goal is to provide accurate assessment of the state of existing software systems in order to assess quality and deploy resources efficiently. The techniques for software developed on the UNIX system use the standard utilities 'find' and 'diff'. Software developed under configuration management by the 'sccs' utility is measured using the 'prs' utility. Techniques are also given for measurement of the amount of reuse in software that was developed on personal computers.

Each of the methods was used for reuse measurement at NASA's Goddard Space Flight Center. The methods were applied to measure reuse in moderately large software systems used for ground centre control of spacecraft.

KEY WORDS: software reuse; software metrics; COTS; configuration management; reuse categories; reuse factors

1. BACKGROUND

Software reuse presents a great opportunity for cost savings in both the software maintenance and development processes. If a portion of a new system's code is obtained from an existing system or from a library of reusable software components, then the resources that ordinarily would have been needed to develop, test and document the reused portion of the system can be saved. Even higher savings are possible if designs, requirements or system architectures are reused from existing systems. In theory, this savings of resources will allow the new system's cost to be reduced.

The true cost of a system depends upon all activities during the software life cycle. Reuse of source code cannot save money in activities such as requirements gathering or software design which generally occur earlier in the software life cycle. The greatest cost savings occur with reuse of higher-level artifacts such as designs, requirements and even entire systems, especially complete, off-the-shelf systems. The cost savings associated with reuse of these so-called 'software artefacts' earlier in the software life cycle is known as 'life cycle leverage'.

CCC 1040–550X/96/050309–12

Received 24 August 1995
Revised 13 June 1996

Unfortunately, the cost savings expected from reuse are not always obtained. That is, a system that claims that 50 per cent of its code was reused will frequently have cost savings that are much smaller.

There are several reasons for such differences between estimates and actual savings:

1. The measurements of the amount of software reused may not be consistent with the information needs of the software managers of the reuse-based project.
2. The evaluation process uses improper definitions of the quantity being measured.
3. There is no appropriate mechanism for collection of measurement data.
4. The true cost of a system depends upon all activities during the software development life cycle (SDLC) and the (sometimes many) following software maintenance life cycles (SMLC). In particular, the amount of testing, integration and software comprehension effort is a large portion of the total life cycle costs, where total life cycle costs are the sum of the cost of the SDLC and all following SMLCs. Proper measurement of the amount of software reuse helps in estimation of the costs of these activities. Improper measurement leads to inefficient use of resources during testing, integration and software comprehension, especially during maintenance.
5. There is an overhead associated with software reuse. This overhead includes development and maintenance of reuse libraries, certification of reusable software components, and searches of the reuse libraries in order to determine appropriate matches with requirements.

In this paper we will focus on the first three issues of reuse cost estimation: proper measurements, definitions and mechanisms for determining the amount of software reuse. In order to simplify the discussion, we will focus on methods of measuring the amount of reuse of source code. Each of the methods discussed in this paper corresponds to a method of measuring the reuse level of higher level software artefacts, such as designs, requirements and even entire systems. The emphasis will be on practical measurement techniques that either fit into an existing metrics framework or can be instituted easily if no such framework exists. Several of the techniques described in this paper were developed because of incompatibilities between the outputs from some existing metrics tools.

The amount of software reuse is used in estimation of testing, integration, and maintenance effort, in general in several ways. A software component that has been certified as to its quality and to its precise interface to other software, whether locally-written or a COTS (commercial off-the-shelf) product, can simplify testing and integration. Maintenance costs should also decrease due to the reduced need for corrective maintenance.

The savings may be even larger for an organization than they are for a particular project. A project that uses an existing software component will save part of its testing, integration and maintenance costs. Other projects using the same reusable software component will have similar savings, resulting in even larger savings for the overall organization. Detailed reuse-based life cycle cost models can be found in Leach (1996, pp. 145–181). Some experiences of Loral Federal Systems with systems that are composed primarily of COTS software are described in Waund (1995).

Many models used for predicting software maintenance costs require accurate measurements of the number of lines of code that are changed, added, or deleted in software that undergoes several revisions. The work of Briand and Basili (1992) is typical.

Unfortunately, this reuse information is not always available directly in computer-

accessible form. Obtaining this information often requires examining many software change reports manually. There is a clear need for some practical approaches to obtaining the amount of reused and new code with minimal human intervention.

A typical cost model for reuse of source code for the classic waterfall process is given below. The constant of 0·125 represents an average overhead for a systematic metrics-driven program of software reuse (Software Engineering Laboratory, 1991, pp. 36–38). The constant of 0·25 represents the most conservative estimate if at least 75 per cent of the existing code is reused. Here the maintenance cost is the sum of the costs of maintaining the reused and non-reused portions of the system.

```
cost = cost of requirements for non-reuse system
       +
       cost to design non-reuse system
       +
       cost to code non-reuse system
       +
       0·125 × (cost to develop non-reuse system)
       +
       0·25 × cost to test non-reuse system
       +
       cost to integrate system
       +
       cost to maintain system
```

A similar reuse-based model of total life cycle costs was developed at Goddard. The model was accurate for software systems with normal integration costs, but underestimated total costs for some systems with unusually complex software interfaces. Currently the model is being extended to include better measurements of system interfaces.

The techniques described in this paper were developed in order to measure the amount of reuse in several relatively large (several hundred thousand lines of code) ground centre software systems used for spacecraft control at NASA's Goddard Space Flight Center in Greenbelt, Maryland. The reference (Mahmot *et al.*, 1994, pp. 1161–1169) describes some of these software systems. The software runs on UNIX workstations, as do most modern ground centre software systems at NASA. (There are a few ground centre software systems for controlling relatively simple spacecraft run on personal computers.)

The techniques described in this paper were developed in the context of our research on flexible, realistic models of total software life cycle cost in the context of a reuse-based software development process. They apply to most software development and maintenance environments.

2. INTRODUCTION

We note that the first step in estimation of anything is to develop proper metrics and a method for their collection and analysis. For software-implemented systems, the metrics should be developed according to the GQM (goals questions metrics) paradigm described in Basili and Rombach (1988).

Several goals are appropriate for estimating the costs and quality associated with a

reuse program that applies to one or more projects. For the 'measure software reuse' goal, appropriate questions might be:

- Are there different types of reuse? If so, how do we measure them?
- How do we determine the size of a system?
- What is the percentage of reuse?
- What is the percentage of reuse in different categories? One organization (Computer Sciences Corporation, 1987) classifies software reuse as being transported, adapted, converted or new. The categories are described in detail in the next section.
- What is the total life cycle cost of the system thus far?
- What is the effect of reuse on the quality of the system?

Appropriate metrics for these questions might be:

- The percentage of reuse.
- The percentage of reuse in the four different categories. (See below for a listing of the reuse categories.)
- The total life cycle cost of the system thus far.
- The defect ratio (= number of software errors divided by the number of lines of code).
- The defect ratio for each of the four different categories.

Each of these metrics requires precise measurement.

In this paper we will show that it is possible to measure the amount of reuse between two software systems in a simple manner. If the software systems are on computers running the UNIX operating system, then standard utilities can be used to simplify the measurement process. Measurement of reuse of software systems on personal computers requires the writing of utility programs to perform the analysis.

3. CATEGORIES OF REUSE

We note that there are several ways in which a software artefact can be reused: totally, in part or not at all. These general groupings can be defined more properly. As an example, there are four categories of reuse specified in a 'standards and practices manual' (Computer Sciences Corporation, 1987) for development of ground system software:

Transported. This means that the code is used as is from a previous system or reuse library. This is also called verbatim reuse.

Adapted. This means that the code is taken from a previous system or a reuse library. The code is subjected to changes, with between 75 per cent and 100 per cent of the code being reused as is.

Converted. This means that the code is taken from a previous system or a reuse library. The code is subjected to more changes than adapted code, with between 50 per cent and 75 per cent of the code being reused as is.

New. The code is considered to be developed from scratch. This category applies if the code is obtained from a previous system or a reuse library and is changed more than 50 per cent. It also applies to code that is developed without making any use of previous systems or code in reuse libraries.

It is clear that each of these categories of software reuse has different impacts on potential cost savings. For example, any code that is reused 100 per cent as is (transported reuse in this terminology) does not need to have any unit testing applied to it. Such testing was assumed to be part of the certification process before the code was put into a reuse library. Source code that is adapted (75–99 per cent reuse) or converted (50–75 per cent reuse) would require some unit testing, although the amount of unit testing might be less than for new code. Source code with less than 50 per cent reuse is generally treated as new code. Different organizations may use different breakpoints for determining the type of reuse in a system. However, the principles remain the same.

We now consider some commonly used definitions that are necessary in order to measure software reuse.

The number of source lines of code, often referred to as SLOC, is one of the most common ways to describe the size of a software system. Proper measurement of it involves a decision about how to count delimiters, lines consisting of both comments and source code, data definitions, included header files, compiler instructions, function prototypes (for C and C++), or package specifications (for Ada).

The following well-known example illustrates the difficulty of measuring lines of C code:

```
#include ⟨stdio.h⟩

main( ){
int i;
for(i = 0; i < 10; i++ )
  {
  printf("%d\n", i);
  }
}
```

Informal experiments with this example often lead to a range of answers for the number of lines of code from one to nine, with seven, five, and three being the most common.

Consider the variation observed when using several tools commonly employed at Goddard for metrics analysis. For the TPOCC (transportable operations control center) system which is used as a reusable software core for use in several spacecraft missions, the lines of code estimate produced for the size of release 1.0 varies from 53 267 to 53 383 to 80 198, depending on the tool used to determine this metric. This is an increase of 50·6 per cent more than the smallest number. Similar results are noted for all other releases of this system.

The term 'line of code' includes everything that affects computation (including data manipulation and decision making), file access or memory allocation. The concept can be refined, as in the way that delimiters, multiple statements on a line, and the contents of included files are handled. Other refinements are possible.

The term 'delivered source instruction', or DSI, includes every arithmetic operation, function call or assignment statement. Delimiters are not counted, nor is the 'include' statement. For the simple code given previously, the DSI metric has the value of four.

The term 'executable source instruction', or ESI, includes only those instructions that affect execution but not memory allocation. Thus statements that use the C language 'malloc( )' function or the C++ or Ada operators 'new' would not be counted as executable

source instructions. Thus the ESI measurement will be smaller than the DSI measurement in most cases. The ESI metric for the simple code given previously has the value of three, since the assignment statement is not counted. There are many other possible counting guidelines.

It should be noted that these metrics all have a close, nearly linear relationship with each other and with any fixed measurement of SLOC. For the TPOCC system mentioned previously, the correlations are all approximately 0·95 or higher. We expect this relationship to hold for most practical measures of lines of code. Thus any of these measurements can be used for analysis of percentage or ratio data, but different measurements using different techniques cannot be compared, since they are absolute numbers. However, any data based on percentage changes to SLOC or on ratios of SLOC, would be valid even with differences in counting methods on sets of projects being compared as long as the definitions were applied consistently.

4. MEASUREMENT OF REUSE PER FILE AND PER FUNCTION

It is much easier to collect and analyse metrics data if we use the file as the primary collection unit. Because many source code files will contain several functions, the data set collected will be smaller and will be easier to visualize. If the existing data on software defects in an organization are kept on a per file basis, then using the file as the primary data source will allow the metrics data results to be easily correlated with the existing discrepancy report data.

If the existing data on software defects are kept on a per function basis, then there are two choices: aggregate the per function data to be consistent with new metrics data collected per file, or separate the new metrics data collected per file into function-specific information. In any software development environment, the objective is to measure reuse in a manner consistent with the way that quality and cost estimation data are organized.

The driving force in metrics collection is the set of questions that we wish to answer. Our eventual goal was to develop reuse-based cost models for the entire software life cycle and to determine the effects of reuse on quality.

The SPA tool from Set Laboratories in Portland, Oregon, was applied to releases 2.0 to 9.0 of the TPOCC software to determine the percentage of reused code. This tool reports information on a per function basis, while software defect data at Goddard are tracked by the file in which the defect was noted. Thus we needed to map data that were collected on a per function basis to data that were collected on a per file basis.

We determined the amount of reuse in a simple manner. For each release of the TPOCC software, we computed a percentage of reuse. The reuse data were taken from a previous analysis using the SPA tool on releases 1.0 to 9.0 of the system. Each function in releases 2.0 to 9.0 was compared with a function of the same name in the previous release.

The reuse factor was obtained by examining the functions in each pair of releases. If there were two functions of the same name in two directories of the same name, then the two functions were assumed to be identical and reuse was considered to be 100 per cent, with no changes. If the function in the later release had a larger number of lines of code than the function of the same name in the earlier version, then we assumed that the changes were minor and used a reuse factor of 90 per cent. If the function in the later release had a smaller number of lines of code than the function of the same name

in the earlier version, then we assumed that the changes were intended to correct a problem and therefore used a reuse factor of 80 per cent. The estimates of 90 per cent and 80 per cent were obtained from reading a small sample of TPOCC source code files and computing the average reuse percentages. Functions whose name appeared in only one of a pair of releases were considered as new code. The individual reuse factors were incorporated into a single reuse factor.

Note that this method provides a general transformation from information collected on a per file basis into information collected on a per function basis. The same transformation technique can apply to any software artefact given entirely in textual format, such as requirements, designs in PDL, test plans, test results and documentation. In each case the first step is to determine files of the same name in the new system and the older system for which you wish to determine reuse. This assumes that there are strict rules for file names, and that reused files keep the same names. Fortunately, this was the case for the source code in the TPOCC system.

From this point on, we will assume that the problem of measurement of software size has been determined satisfactorily, and that the units chosen for reuse measurement are consistent with the organization's method of collection of quality and cost measurements. (Usually the choice is between the use of files or functions, as was discussed previously.) In the remainder of this paper, we consider the problem of measurement of reuse in detail, with special emphasis on the estimation of future (normally maintenance) life cycle costs after a system has been created using some reusable software components.

5. MEASUREMENT OF REUSE ON UNIX SYSTEMS USING diff AND find

It is easy to measure differences between source code files on UNIX systems. For simplicity, we assume that both the original code and the new code exist on the same computer. The extension of these ideas to software that resides on different nodes in a network is straightforward and will be omitted.

The technique is to use the UNIX 'diff' utility which measures the difference between files. The two files used as arguments to the 'diff' utility are compared line by line, producing as output the smallest number of lines that have to be changed in order to make the two files identical. Thus, if two files differ by the insertion of one line in a file, 'diff' reports a single line of difference between the two files, rather than a cascading sequence of changes to all subsequent lines after the insertion.

The result of the 'diff' operation can then be piped to the UNIX 'wc' utility in order to determine the number of changed lines. This is illustrated by the combined command that prints only the number of lines in the difference of the two files:

```
diff FILE_OLD FILE_NEW | wc -l
```

The ratio of this number to the size of the file FILE_NEW represents the proportion of this file that is new and not obtained from reuse of the file FILE_OLD. The number of lines in a file can be found by using the 'wc' command as in

```
wc FILE_NEW
```

If the two files have the same name, but are in different directories, then the ratio is the percentage of reuse of the file. As such, the level of reuse can be determined as being transported, adapted, converted or new, in the previously used terminology.

The results can be aggregated by writing them to an output file together with the names of the directories and subdirectories in which the compared files occur. If the projects were developed according to strict naming conventions, as was the case with the software system we described previously, the higher level directories will represent particular subsystems that can be compared easily.

The 'diff' command requires two arguments, which are the names of files to be compared. It can be invoked directly on a pair of files. It is more common to use 'diff' together with the powerful 'find' utility in order to search directories recursively.

An example of the use of 'find' is given in the following algorithm:

1. REUSED_FILE = FILE_OLD
2. findNEW_DIR-nameREUSED_FILE-print|diffREUSED_FILE
3. Repeat step 1 for each existing older system for which you wish to determine reuse. Use the 'diff', 'find', and 'wc' commands as before.
4. Determine the number of files that are not reused. These files are considered new.
5. Determine the number of lines of code of each file using the 'wc' command as before and compute the total number of lines of code of each file.

Unfortunately, the 'diff' tool cannot measure lines of code directly. However, it does provide some useful information.

We note that it is easy to compute a more refined measurement than just the number of lines in a file. If more detailed information about the number of lines of code in a file is needed, then a program must be written in order to determine this quantity. The easiest way to do this is to use a lexical description of the source code language and the UNIX 'lex' and 'yacc' tools in order to develop a lexical analysis tool to determine differences in the lines of code (LOC, DSI, ESI, or similar).

A user-defined utility to compute the number of lines of code can be inserted in appropriate places in the shell commands given above. The interface for this utility is easy to write because of the UNIX standard shell convention of allowing standard input and output of processes to be linked by pipes using the '|' symbol.

6. MEASUREMENT OF REUSE ON UNIX SYSTEMS USING CONFIGURATION MANAGEMENT

There is an additional source of reuse information if the software was developed under configuration management, or CM. Most implementations of configuration management begin by making a copy of each file placed under CM. They then store each version of a file under CM as a set of deltas, which are changes to the base file. It is thus possible to recover any file from this information. The advantage of using CM techniques for measuring software reuse is that there is no need to have strict conventions for naming files, as was the case with the previous measurements using 'diff' and 'find'. Using CM techniques for reuse measurement also allows multiple versions of an evolving reusable software system to be evaluated for reuse in one step, rather than by the pairwise comparison that was described in the previous section.

A common UNIX implementation of CM is the 'sccs' utility. The version of 'sccs' on the HP-UX operating system includes a utility called 'prs', which can print the number of changes made to a file, the number of changes of lines within a file, and the number of such changes within any given time period.

The 'sccs' utility has the following features:

- A directory named SCCS is created when the 'sccs' utility is invoked for the first time.
- Copies of the source code files are placed in the SCCS directory.
- These copies are stored in ASCII format as a base file together with changes to the file. These changes are called deltas.
- Only one copy of a file in the SCCS directory can be changed at a time.
- Explicit action must be taken to examine or change a file under sccs control.

The sccs tool allows any files under its control to be rolled back to a previous state if a change was found to be undesirable. It does not provide statistics on the number and type of changes. The statistical information can be obtained by using a companion tool called 'prs'. That utility can be used in several ways. The single command

```
prs -d"Li:\t:Ld:\t:Lu" file_name
```

prints three values on a line, separated by tabs: the number of lines inserted, the number of lines deleted and the number of lines that are unchanged. The number of changed lines can then be computed as a percentage of the total number of lines.

Unfortunately, the 'prs' tool evaluates changes in all lines in the file (including comment lines) and therefore cannot determine only those changes to actual lines of code. However, estimates on the percentage of changes in the total number of lines are likely to be highly correlated with the actual number of lines of code, or some other measure such as DSI or ESI. The correlation coefficients were above 0·97 for the systems that we considered. Thus, all our methods of measuring software size were satisfactory the predictors of the amount of new testing and maintenance effort when used with the 'prs' tool for measuring software reuse.

7. MEASUREMENT OF SOURCE CODE ON PERSONAL COMPUTERS

There are several MS-DOS commands that are related to the UNIX utilities in functionality: DIR and COMP. However, one of the major measurement problems is the lack of interoperability between such utilities. This caused some difficulty when measuring reuse of source code on computers running MS-DOS or Windows.

The DOS command DIR will recursively search a directory that is specified by its argument if the /S option is used with the command as in

```
DIR/S NEW_DIR
```

Unfortunately, this is not as clean as the use of the UNIX 'find' command, since DIR

produces as output the name of the directory or subdirectory being searched at the moment, followed by the name of the file within the directory. That is, the DOS command DIR produces only the name of the file as seen within the directory, not the complete path name. Thus, some sort of post-processing must be performed in order to be able to distinguish two different files with the same relative path names, but different absolute path names.

The COMP command is even less helpful. This command has one of two outputs when given two files named FILE1 and FILE2 as arguments:

```
Comparing FILE1 and FILE2 ...
Files compare OK
Compare more files (Y/N) ?
```

or

```
Comparing FILE1 and FILE2 ...
Files differ
Compare more files (Y/N) ?
```

A flag can be set using the N option to print the number of characters that differ between the files. However, the output is so large in this case as to be meaningless. In each case the user must respond to the prompt and indicate if any additional processing is needed.

Because of these limitations of the available DOS commands, there are essentially only three ways to measure the percentage of reuse for two different systems:

- Write utility programs to measure the amount of reuse between two files and combine the measurements of pairs of files with another utility program to obtain aggregate information of the two systems.
- Transfer the MS-DOS-based software to a UNIX system and use the standard UNIX utilities as before.
- Use commercially available programs to compute the percentages of modified versus unchanged code.

The second method is probably preferable, since we can reuse existing utilities without any new programming.

8. MEASUREMENT OF REUSE OF OTHER SOFTWARE ARTEFACTS

Measuring the amount of reuse in higher level software artefacts is more difficult than measuring reuse of source code. The wide range of design representations (text-based PDL, graphics-based flowcharts, data flow diagrams, or other diagrams CASE tools, etc.) make automatic collection of reuse information difficult. We will therefore be content with a description of the measurement process.

Requirements are typically given in textual format, with the individual requirements

listed as bulleted or numbered items. The requirements are often presented in outline form. Thus a simple measure of the amount of requirements reused between two systems is the percentage of requirements reused. For a very small system, this can be done by a hand count of the number of requirements in each.

A more elaborate estimate can be based on the number of function points in the requirements (Drummond, 1992). The term 'function point' is used in the sense of Albrecht (1979), who first popularized the use of function point metrics early in the software life cycle. The International Function Point User Group publishes a periodically updated guide to counting function points (Abran and Desharnais, 1995). Counting either requirements or function points can be done by creating a spreadsheet from two existing requirements traceability matrices for the two systems being compared for reuse.

The amount of reuse in two designs can be computed in the same manner as requirements reuse if the designs are given in textual form, such as pseudo-code or a PDL (program design language). There is little that can be done yet to automate measurement of the amount of reuse of designs that are given in graphics format.

The same type of analysis can be applied to reuse of text-based documentation, test plans and test results.

9. CONCLUSIONS AND FUTURE WORK

It is possible to measure the amount of reuse between two systems using simple tools that fit into an existing metrics program. It is even possible to measure the amount of reuse in different releases of the same software system. These percentages of reuse can be used in simple cost models to predict the cost of life cycle activities such as testing, integration and maintenance effort. We intend to pursue the inclusion of metrics for software interface complexity in general reuse-based life cycle software cost models in a future paper.

Measurement is easier if the software is developed according to strict naming conditions. Two standard UNIX utilities, 'find' and 'diff', can be used easily to provide good estimates of the amount of reuse. Additional information can be found if the software was developed under the UNIX configuration management tool 'sccs'.

Many of the techniques given in this paper also apply to measurement of the amount of reuse in either requirements or non-graphical designs. We are not aware of any general techniques that can automate measurement of the amount of reuse in designs given in graphical format or in any representation that is not text-based.

All the methods given in this paper are automated easily and can be applied with little overhead. The methods are especially useful if the amount of reuse was not obtained before the project started.

Acknowledgements

Some of this research was performed while the author was a NASA-ASEE Summer Faculty Fellow at the Goddard Space Flight Center in Greenbelt, Maryland. Other parts of the research were partially supported during the academic year by NASA grant numbers NAG-5–2904 and NAG-5–3156 and by the US Air Force Office of Scientific Research under grant number 49620–95–1–0526.

The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Office of Scientific Research or the U.S. Government.

Many thanks to Judith Bruner, Jack Koslosky, Henry Murray and Kelly Jeletic of Goddard for helpful conversations and data access. Many thanks also to Linda Rosenberg of Unisys for running the SPA tool on the different TPOCC releases. Finally, special thanks are due the anonymous referees of the *Journal* for improving this paper.

References

Abran, A. and Desharnais, J.-M. (1995) 'Measurement of functional reuse in maintenance', *Journal of Software Maintenance*, **7**(4), 263–277.

Albrecht, A. J. (1979) 'Measuring application development productivity', in *Proceedings of the IBM Applications Development Joint SHARE/GUIDE Symposium*, SHARE, Chicago, IL, pp. 83–92.

Basili, V. R. and Rombach, H. D. (1988) 'The TAME project: towards improvement-oriented software development', *Transactions on Software Engineering*, **SE-14**(6), 758–773.

Briand, L. C. and Basili, V. R. (1992) 'A classification procedure for the effective managment of change during the maintenance process', in *Proceedings of the Conference on Software Maintenance—1992*, Orlando, Florida, IEEE Computer Society Press, Los Alamitos, CA, pp. 328–336.

Computer Sciences Corporation (1987) *Standards and Practices Manual for the SEAS Project Standard Software Development Methodology*, Computer Sciences Corporation, Laurel, MD.

Drummond, I. (1992) *Estimating with MkII Function Point Analysis*, HMSO, London.

Leach, R. J. (1996) *Software Reuse: Methods, Models, Costs*, McGraw-Hill, New York, NY.

Mahmot, R., Koslosky, J., Beach, E. and Schwarz, B. (1994) 'Transportable payload operations control centre reusable software: building blocks for quality ground data systems', in *Proceedings of the Third International Symposium on Space Mission Operations and Ground Data Systems*, Greenbelt, MD, NASA Publication 3281, Vol. 2, NASA Goddard Space Flight Center, Greenbelt, MD, pp. 1161–1169.

Software Engineering Laboratory, (1991) *Proceedings of the Sixteenth Annual NASA/Goddard Software Engineering Workshop: Experiments in Software Engineering*, NASA/Goddard Space Flight Center, Greenbelt, MD.

Waund, C. (1995) 'COTS integration and support model', in *Systems Engineering in the Global Marketplace: Proceedings of the NCOSE International Symposium*, St. Louis, Missouri, NCOSE, Washington, DC, pp. 205–212.

Author's biography:

Ronald J. Leach is a Professor in the Department of Systems and Computer Science at Howard University. He does research in several areas of software engineering, including reuse, fault-tolerance, performance modelling and process improvement. He received the BS, MS and Ph.D. degrees in Mathematics from the University of Maryland, and an MS in Computer Science from Johns Hopkins. Ronald is the author or co-author of four books and over 50 technical papers. He can be reached via e-mail at rjl@scs.howard.edu.